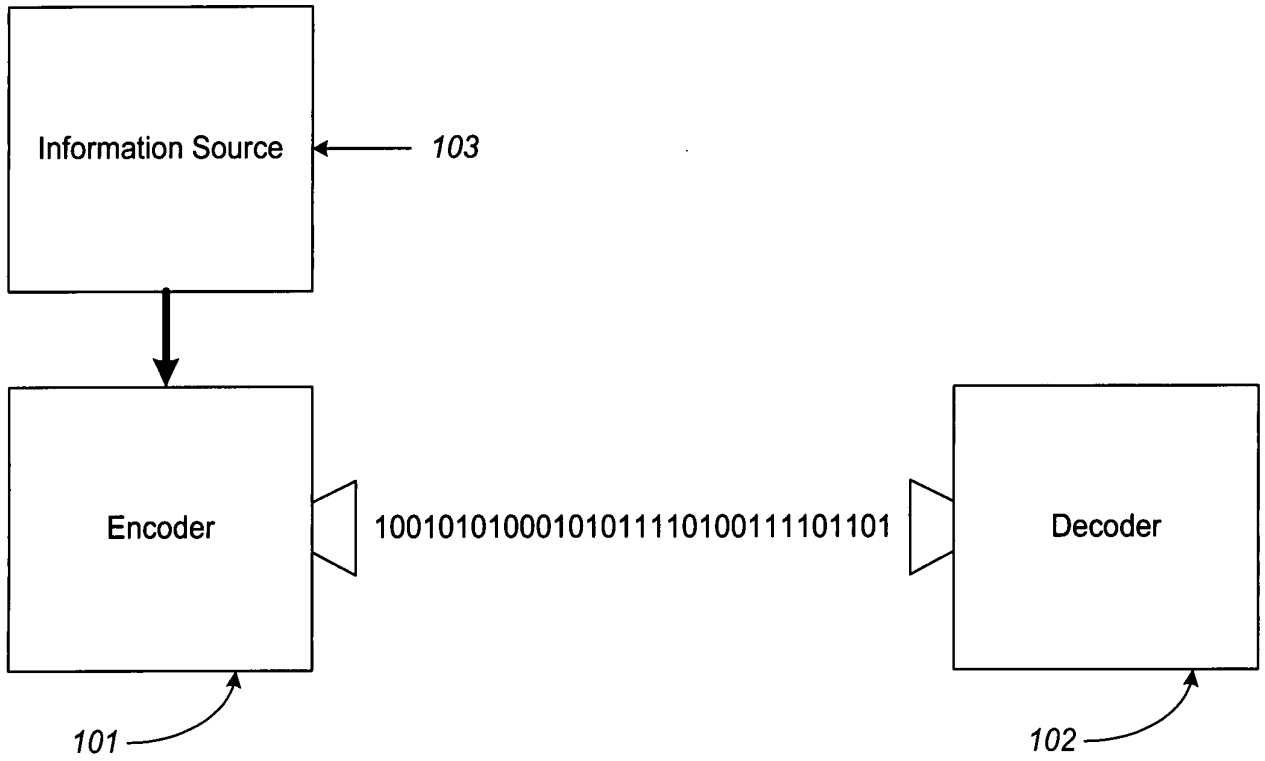

Information Source
103
Encoder
10010101000101011110100111101101
Decoder
101
102

FIG. 1

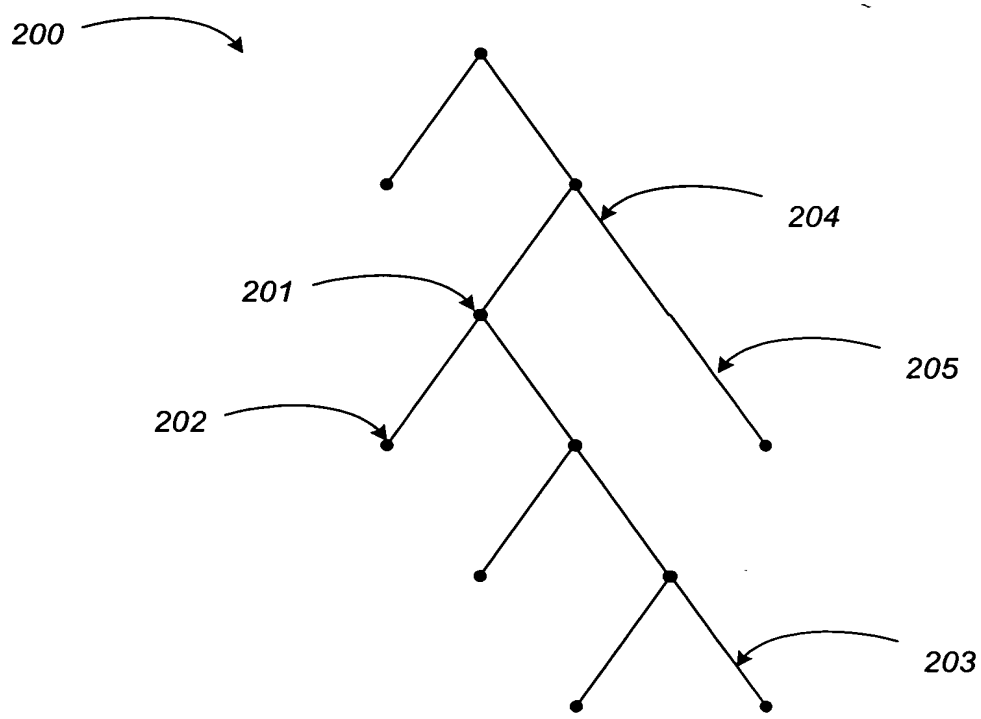
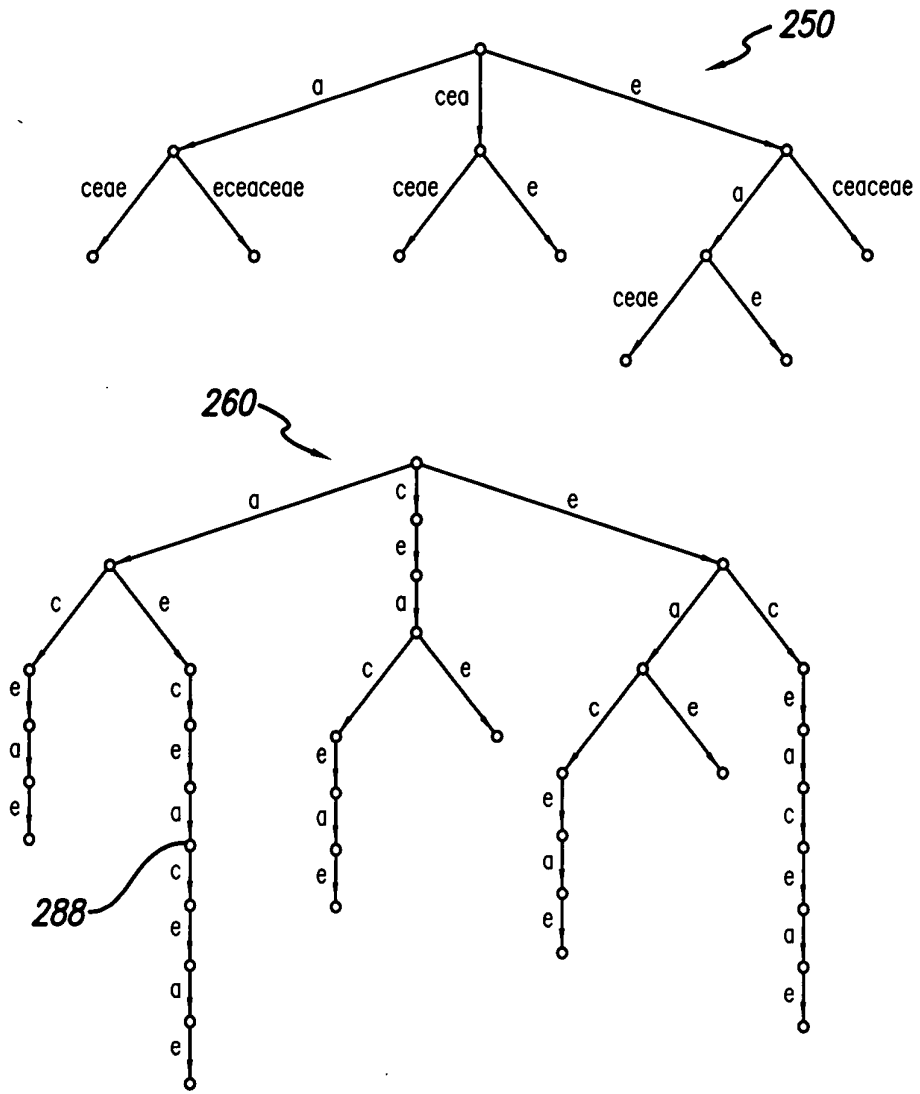


FIG. 2A

FIG. 2B


250
a
cea
e
ceae
eceaceae
ceae
e
a
ceaceae
ceae
e
260
a
c
e
c
e
a
c
e
e
a
e
c
e
a
c
e
e
a
e
c
e
a
c
e
e
a
e
c
e
a
c
e
e
a
e
c
e
a
c
e
e
a
e
288

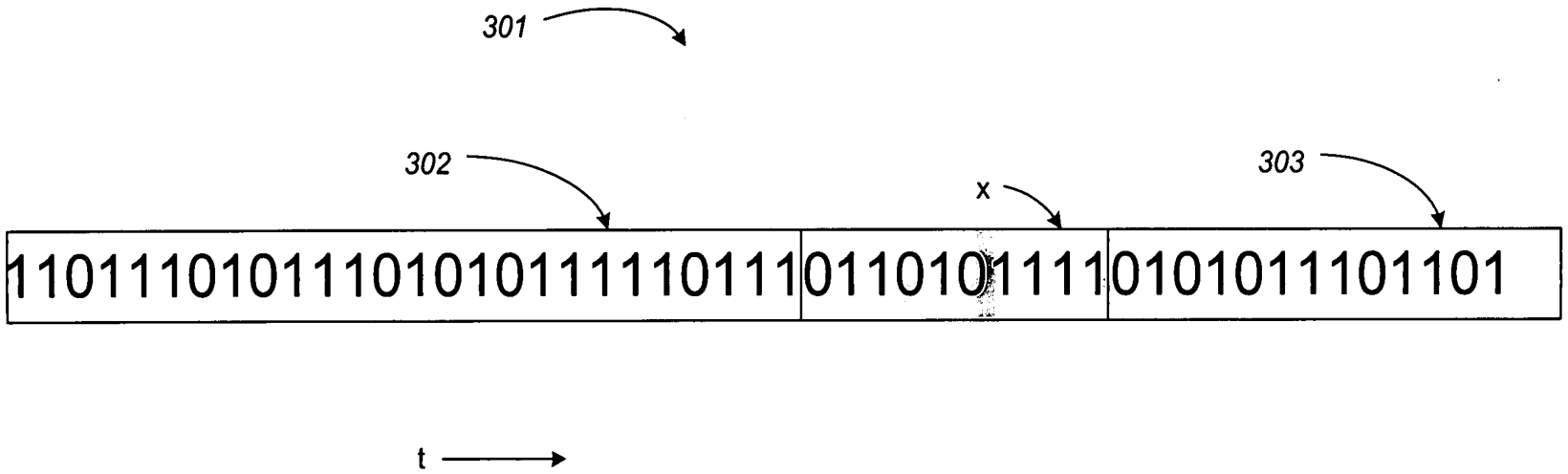

301
302
x
303
1101110101110101011111011101101011111010101101101
t

FIG. 3

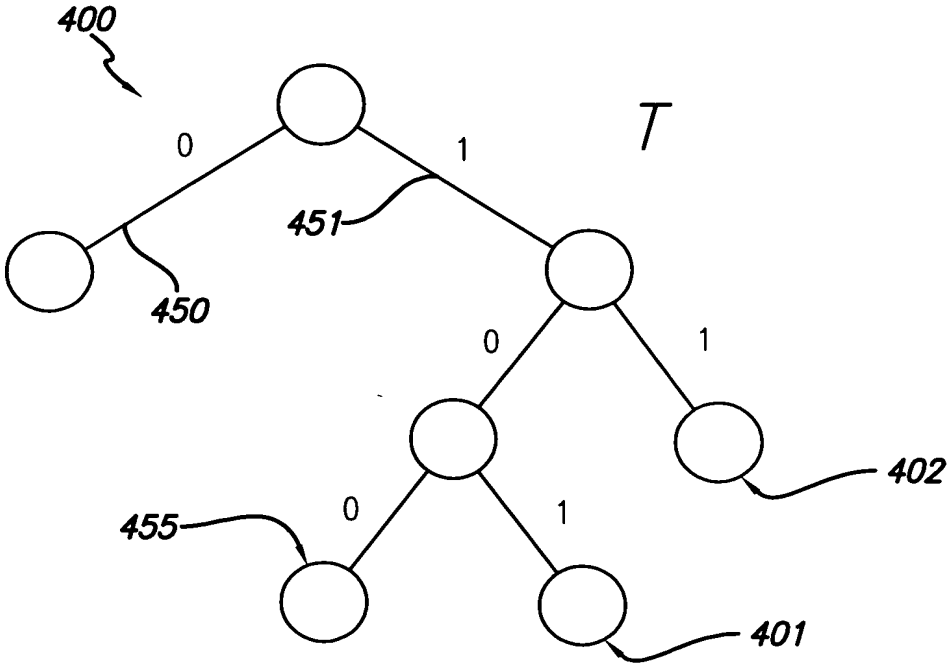


FIG. 4

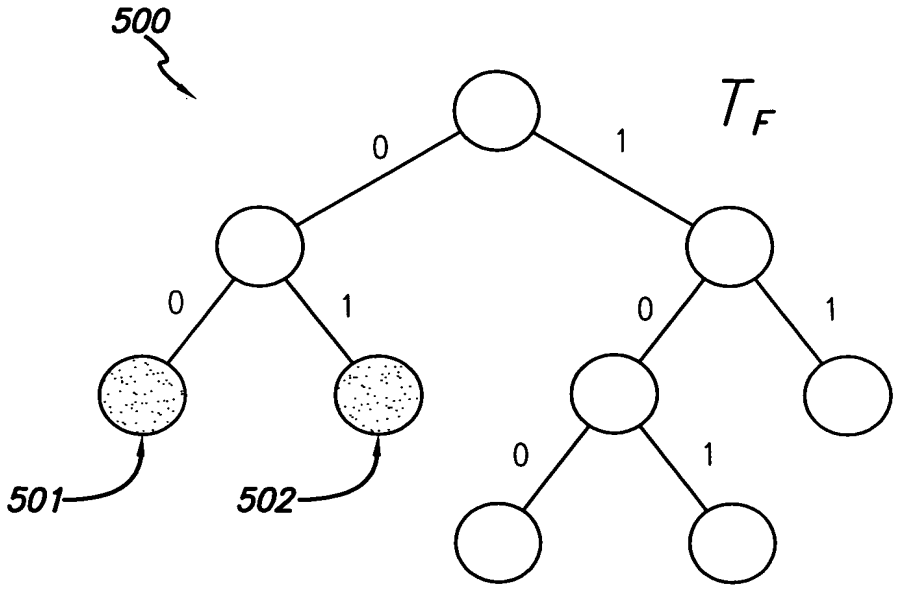


FIG. 5

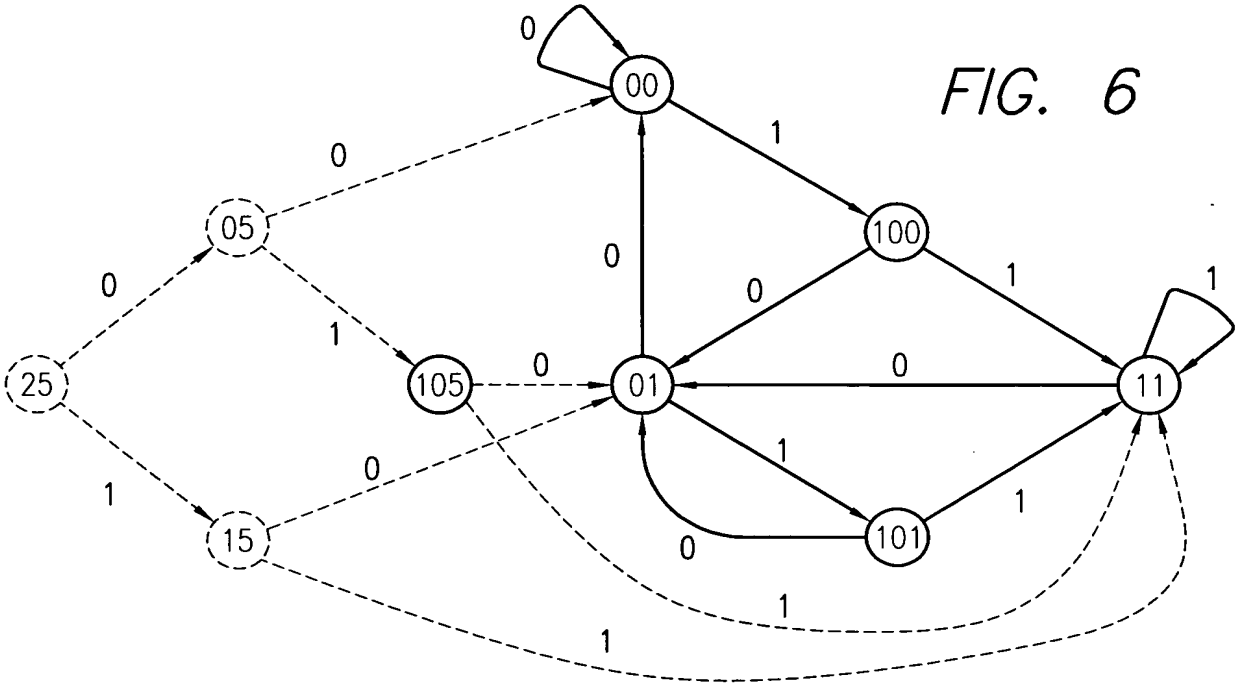


FIG. 6

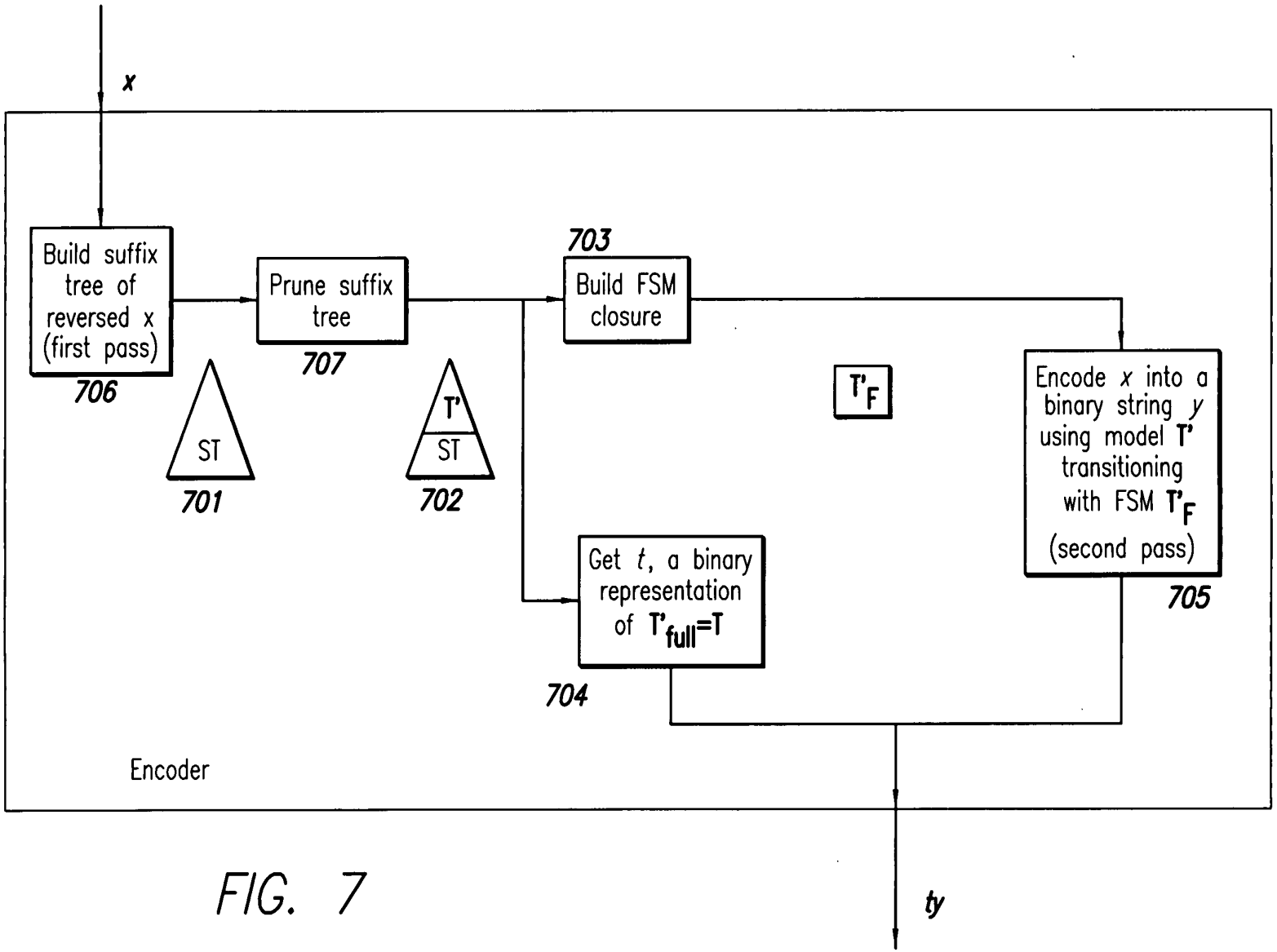

x
Build suffix tree of reversed x (first pass)
706
Prune suffix tree
707
703
Build FSM closure
ST
701
T'
ST
702
T'F
Encode x into a binary string y using model T' transitioning with FSM T'F (second pass)
705
Get t, a binary representation of T'full=T
704
Encoder
ty

FIG. 7

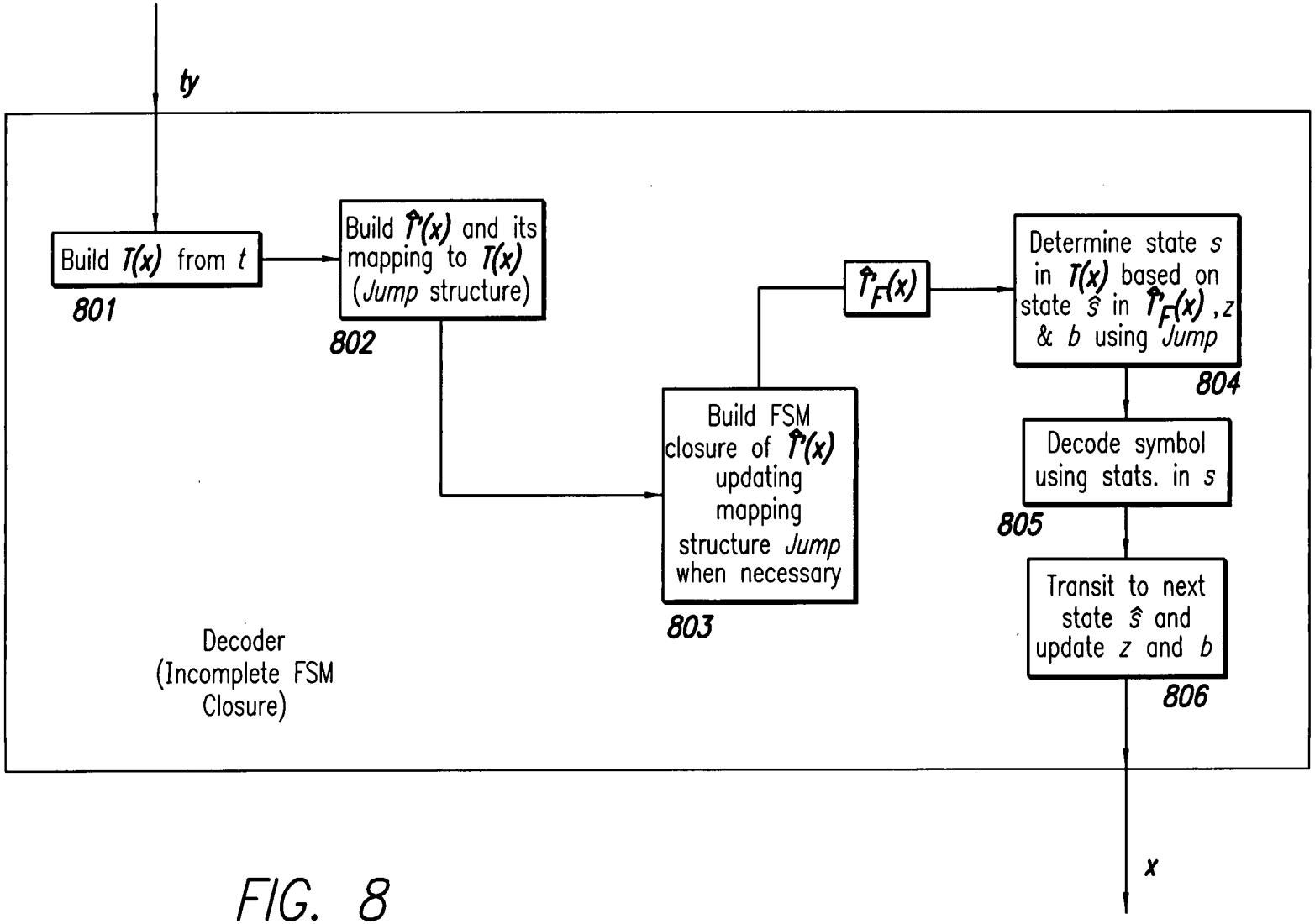

ty
Build T(x) from t
801
Build T̂'(x) and its mapping to T(x) (Jump structure)
802
Build FSM closure of T̂'(x) updating mapping structure Jump when necessary
803
T̂'F(x)
Determine state s in T(x) based on state ŝ in T̂'F(x), z & b using Jump
804
Decode symbol using stats. in s
805
Transit to next state ŝ and update z and b
806
x
Decoder (Incomplete FSM Closure)

FIG. 8

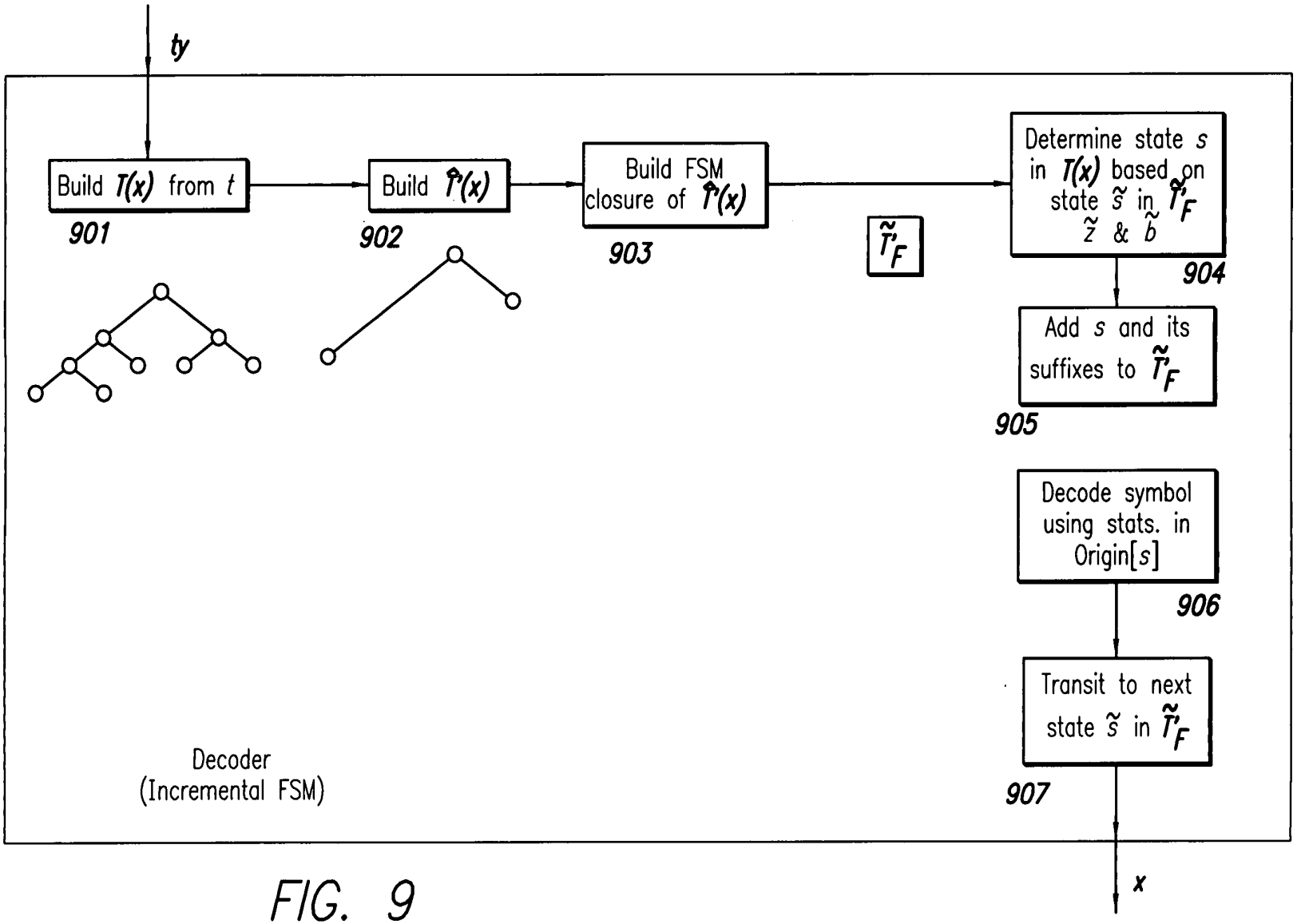

ty
Build T(x) from t
901
Build T'(x)
902
Build FSM closure of T'(x)
903
T'F
Determine state s in T(x) based on state s̃ in T'F z̃ & b̃
904
Add s and its suffixes to T'F
905
Decode symbol using stats. in Origin[s]
906
Transit to next state s̃ in T'F
907
x
Decoder
(Incremental FSM)

FIG. 9

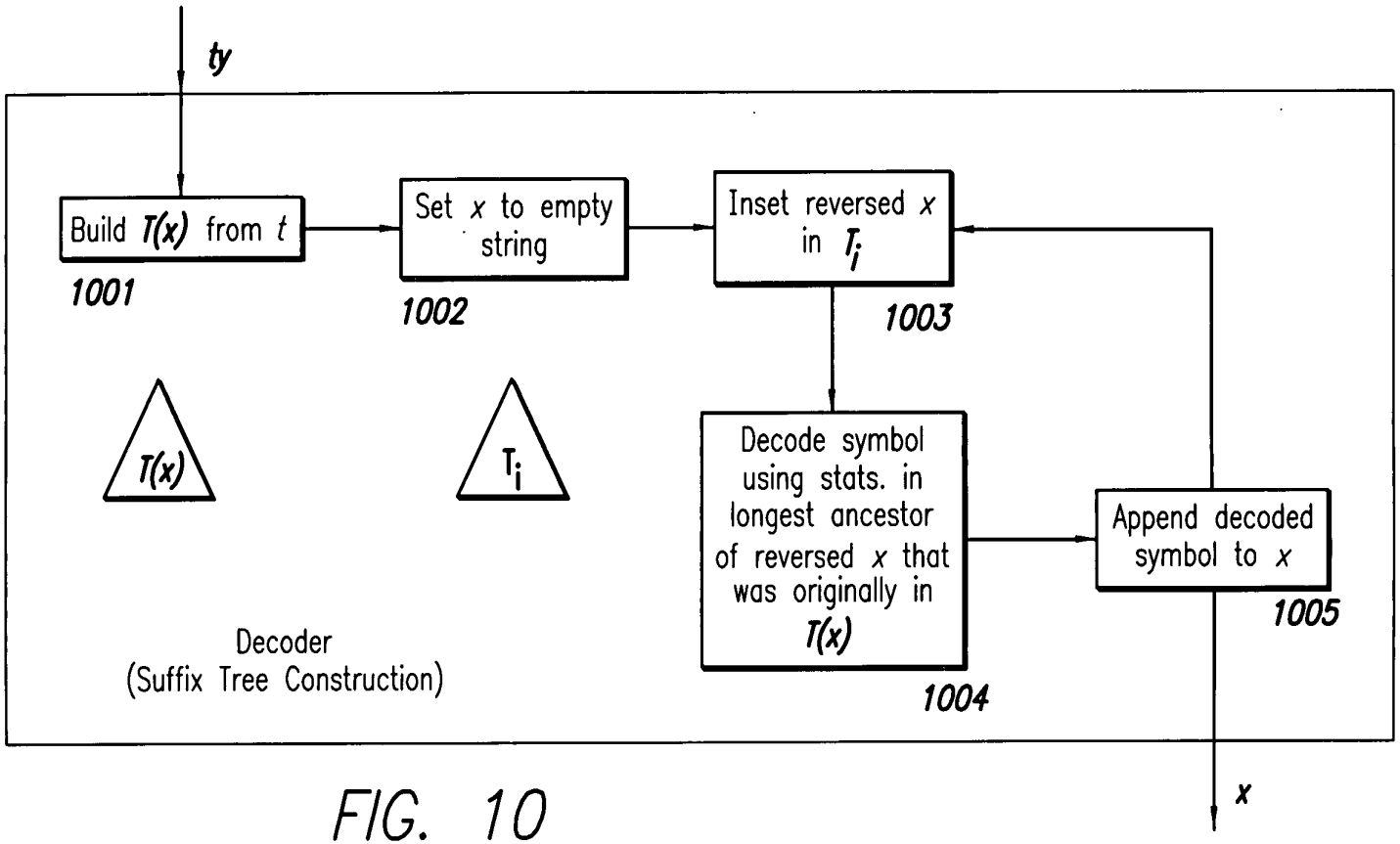

ty
Build T(x) from t
1001
Set x to empty string
1002
Inset reversed x in T_i
1003
T(x)
T_i
Decode symbol using stats. in longest ancestor of reversed x that was originally in T(x)
1004
Append decoded symbol to x
1005
Decoder
(Suffix Tree Construction)
x

FIG. 10

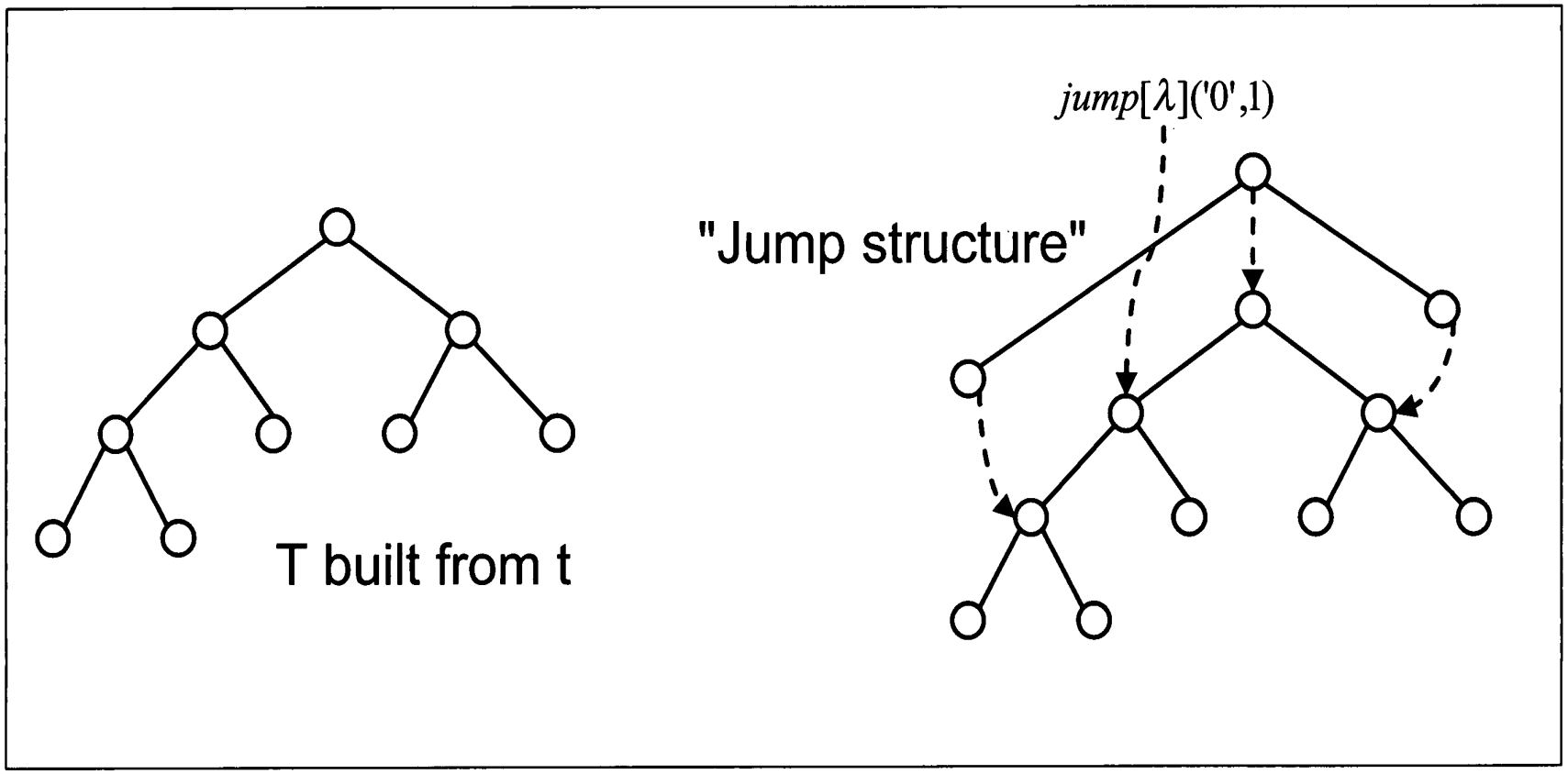


FIG. 11

FIG. 12

```
1. Set T̂′(x) = compact (T(x) − leaves(T(x))
2. Compute T̂′F(x), FSM closure of T̂′(x)
3. Set ŝ = λ //Pointer to root node
4. Set zlength = 0 and b = λ
5. while not end of input
6.    if zlength > 0
7.       Determine head(z) using symbols decoded so far
8.       if ŝ is an internal node of T(x) and head(z) ∈ A, set s = ŝzb
9.       else if ŝ is an internal node of T(x) and head(z) ∉ Aŝ, set s = ŝhead (z)
10.      else if ŝ is a leaf of T(x), set s = ŝ
11.      else set s = Origin [ŝ]
12.      //Note: in any case s is a pointer to a node in T(x)
13.   else
14.      if ŝ is a node of T(x), set s = ŝ
15.      else set s = Origin[ŝ]
16.   end if
17.   Decode next symbol using statistics in s
18.   Update statistics in s
19.   Set ŝ to next state in TF according to the decoded symbol
20.   Update values of zlength and b
21.end while
```

Decoding Using Incomplete FSM closure

```
1.  Set T̂'(x) = compact(T(x) − leaves(T(x)))
2.  Compute T̃'F, FSM closure of T'(x)
3.  Set s̃ = λ //Pointer to root node
4.  Set z̃length = 0 and b̃ = λ
5.  while not end of input
6.      if z̃length > 0
7.          Determine head (z̃) and symbols decoded so far
8.          Create node s̃z̃ splitting edge departing from s̃ which first symbol is head(z̃)
9.          Set r = s̃z̃
10.         Set Transitions[r] = Transitions[s̃]
11.         Verify*(r)
12.     else
13.         Set r = s
14.     end if
15.     if b̃ ≠ λ
16.         Create node rb̃
17.         Set Transitions[rb̃] = Transitions[r]
18.         Verify*(rb̃)
19.         Set s = rb̃
20.     else
21.         Set s = r
22.     end if
23.     Decode next symbol using statistics in Origin[s]
24.     Update statistics in Origin[s]
25.     Set s̃ to next state in T̃'F according to the decoded symbol
26.     Update values of z̃length and b̃
27. end while
```

Decoding Using Incremental FSM closure

FIG. 13

FIG. 14

Decoding Using Incremental Suffix Tree Construction

```
1. Initialize short-cut links for T(x)
2. Set r' = λ and s = λ
3. while not end of input
4.     Decode x_i using statistics in s
5.     //Upwards traversal
6.     Set v = r'
7.     while v ≠ λ and v has no short-cut link of v for x_i
8.         Set v = PARENT(v)
9.     end while
10.    if v has a short-cut link for x_i
11.        Set w - node pointed by short-cut link of v for x_i
12.    else
13.        Set w = λ
14.    end if
15.    if |w| > |v| + 1
16.        Split edge from PARENT(w) to w inserting node x_i v
17.        Set r_new = x_i v
18.        Set u = v
19.        while short-cut of u for x_i = w
20.            Set short-cut link of u for x_i pointing to r_new
21.            if u ≠ λ, set u = PARENT(u)
22.        end while
23.    else
24.        //Downwards traversal
25.        if Jump[v] defines a mapping for x_i
26.            Set r_new = last entrance of Jump[v] for x_i
27.        else
28.            Set r_new = w
29.            Set j = |r_new|
30.            while i - j > 0 and r_new has a child in the direction of x_i - j
31.                Set r_new = child of r_new in the direction of x_i-j
32.                Update Jump [v]
33.                Increment j
34.            end while
35.        end if
36.    end if
37.    Add child to r_new representing suffix x̄^i
38.    For all nodes in the path from r' to v (excluded)
39.        Set short-cut link for symbol x_i pointing to the new node x̄^i
40.    end for
41.    Set r' = r_new
42.    Set s = longest prefix of x̄^i that was originally in T(x)
43. end while
```